CS5003 Data Structures and Specialist Programming

Weighting: 30% of the module mark

Deadline: Friday 13th of December 2024

Module Leader: Mona Abdelgayed ***Student ID: 23032697***

Contents

Detailed Instructions to Run the Program

- Running the Program and installing JDK - The beginning stage of the coursework shows the launch of the application. It provides precise instructions to ensure clarity and illustrate the project's beginning point. Installing the JDK kit is necessary in order to configure the code and classes for operation.

- Downloading IDE - Installing an IDE, like Eclipse or IntelliJ IDEA, facilitates development by providing features for project management, code navigation, and debugging.

- Download source files here – This helps verify that every .java file is accessible. By listing the project's files as Main.java, BankAccount.java, etc.

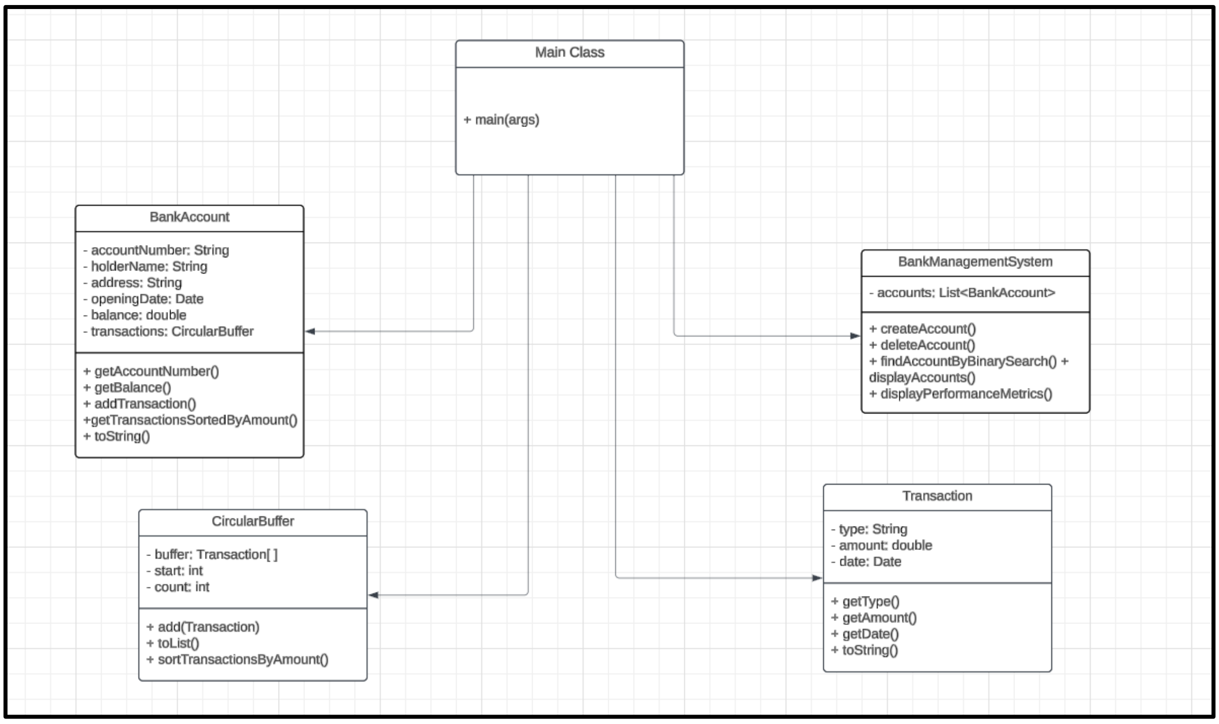
- Opening Project and adding files - Arranging files, creating and launching a new project in the selected IDE. This guarantees that dependencies are fixed, include all.java files in the project.

- Compile and execute the program - Looking for and correcting any mistakes. The setup is confirmed in this stage. Then the file includes the entry point (main method), which starts the application when it starts up.

- Use and Interact - Test features and in this case, adding transactions and establishing accounts using the menu-based console interface.

Architecture of the Software

UML Diagram of Java code:



Detailed Description of Classes, Data Structures, and Algorithms

Transaction Class:

A bank account transaction is represented by the Transaction class.

Fields:

(String type): The type of transaction, e.g., "withdrawal" or "deposit".

(double amount): The amount involved in the transaction.

(Date date): The date the transaction occurred, initialized to the current date and time when the transaction is created.

Constructor :

Takes (type) and (amount) as parameters and initializes the respective fields.

Sets the (date) field to the current date and time.

Methods :

getType(): Returns the type of the transaction.

getAmount(): Returns the amount of the transaction.

getDate(): Returns the date of the transaction.

toString(): Returns a string representation of the transaction in the format:

"Type: [type], Amount: [amount], Date: [date]".

The Transaction class represents individual financial transactions and contains the type (e.g., "deposit" or "withdrawal"), amount, and date of the transaction. By guaranteeing structured data processing and eliminating transaction-specific code from the BankAccount class, this enhances modularity. The Date field enables accurate monitoring, but methods such as getType, getAmount, and toString provide limited access and ready-to-read output. This design enhances the financial system's scalability, maintainability, and clarity.

CircularBuffer Class:

The CircularBuffer class is used to store and manage a fixed-size collection of Transaction objects, where older transactions are overwritten by newer ones once the buffer is full.

Fields :

Transaction[] buffer: An array of Transaction objects that stores the buffer data.

int start: The index of the oldest transaction in the buffer.

int count: The number of transactions currently in the buffer.

Constructor :

Takes a size parameter to initialize the buffer with the given size.

Methods:

add(Transaction transaction): Adds a new transaction to the buffer. If the buffer is full, the oldest transaction is overwritten.

toList(): Converts the circular buffer into a List<Transaction>, starting from the oldest transaction.

sortTransactionsByAmount(): Sorts the transactions in the buffer by the amount in descending order. It uses insertion sort, which is applied to the list of transactions before updating the buffer.

A fixed-size collection of transactions may be managed effectively with the CircularBuffer class, which makes sure that only the most recent entries are kept. To maximise memory utilisation and speed, it employs an array-based circular buffer, which automatically overwrites the oldest data as the buffer fills up. This approach guarantees real-time access to recent transactions while preventing excessive memory development. The banking system's functionality and speed are improved while keeping things simple thanks to methods like add, toList, and sortTransactionsByAmount, which make it easy to add, retrieve, and sort transactions.

BankManagementSystem Class

Manages the creation, deletion, and retrieval of bank accounts.

Properties:

accounts (List<BankAccount>): Stores all the accounts in the system.

Methods:

createAccount(String accountNumber, String holderName, String address, double initialBalance): Creates a new account and ensures valid input (e.g., 8-digit account number, unique identifier).

deleteAccount(String accountNumber): Deletes an account using binary search to locate it.

findAccountByBinarySearch(String accountNumber): Searches for an account using binary search for efficiency.

displayAccounts(): Lists all accounts in the system.

displayPerformanceMetrics(): Displays system performance metrics.

Data Structures Used:

List<BankAccount> to store accounts.

Algorithms:

Binary Search: Used in findAccountByBinarySearch() to efficiently locate accounts in a sorted list.

Comparator-based Sorting: Ensures accounts are sorted by account number.

The application's central component, the BankManagementSystem class, controls the addition, removal, and arrangement of bank accounts. It allows for dynamic scalability by storing accounts in an ArrayList. For effective account retrieval, the class uses binary search, guaranteeing speedy access even when there are many accounts. It offers ways to view account data, deactivate accounts, and establish accounts with validation. The class facilitates scalability, simplifies operations, and maintains system responsiveness and organisation by centralising account administration.

Main Class

Provides a console-based user interface for interacting with the system.

Methods:

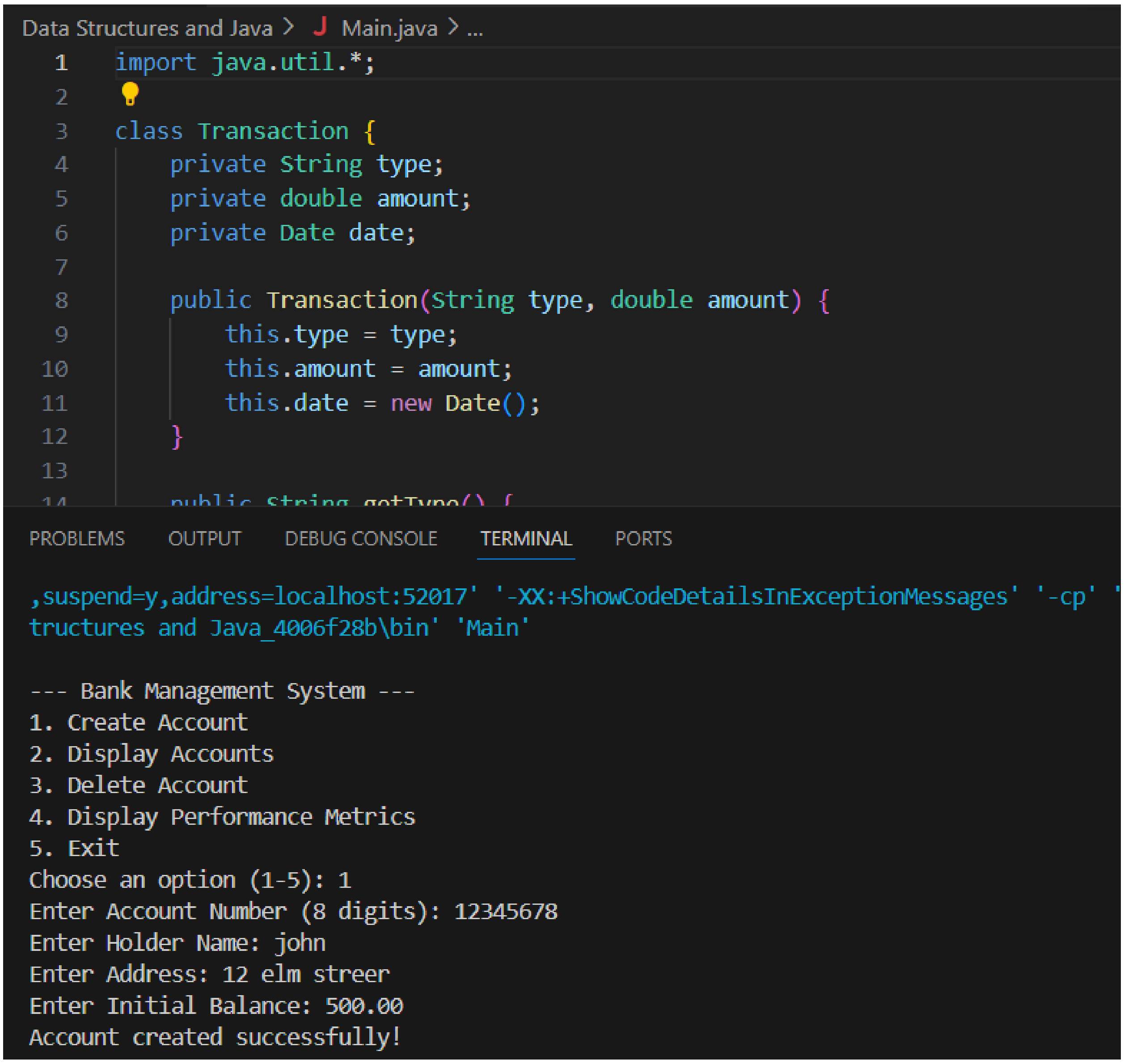
Contains a main menu for creating accounts, displaying accounts, deleting accounts, viewing performance metrics, and exiting the application.

Data Structures Used: Scanner for user input.

The Main class serves as the application's entry point and offers a user-friendly interface for interacting with the system. To make account creation, deletion, display, and performance measurements easier, it uses a console-based menu. It connects the user to the core features of the BankManagementSystem by processing user input. The Main class guarantees smooth system interaction by assisting users at every stage and managing program flow and error handling.

Screenshots of program running

This function allows the user to input the necessary details (account number, holder name, address, initial balance) to create a new bank account. It validates the inputs and ensures no duplicate account numbers exist.

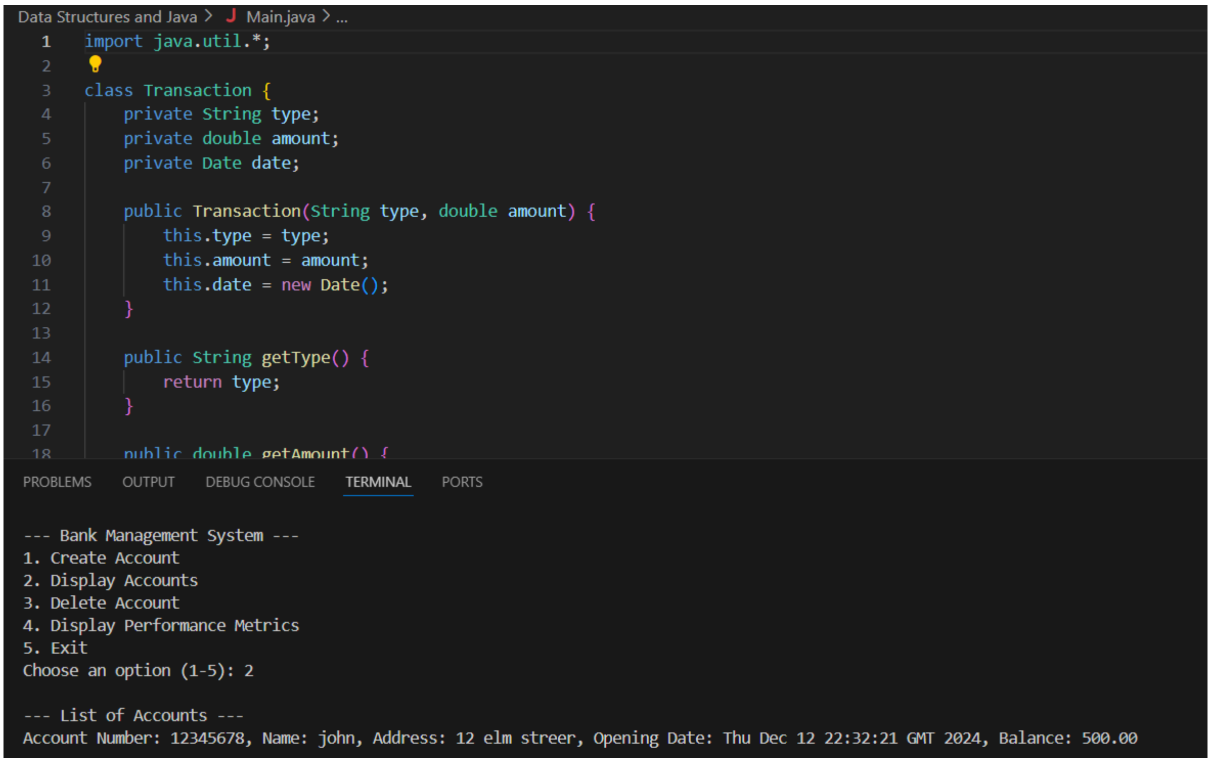


Data Structures and Java > Main.java > ...

```java
import java.util.*;

class Transaction {
    private String type;
    private double amount;
    private Date date;

    public Transaction(String type, double amount) {
        this.type = type;
        this.amount = amount;
        this.date = new Date();
    }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
,suspend=y,address=localhost:52017' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' '
tructures and Java_4006f28b\bin' 'Main'

--- Bank Management System ---
1. Create Account
2. Display Accounts
3. Delete Account
4. Display Performance Metrics
5. Exit
Choose an option (1-5): 1
Enter Account Number (8 digits): 12345678
Enter Holder Name: john
Enter Address: 12 elm streer
Enter Initial Balance: 500.00
Account created successfully!
```

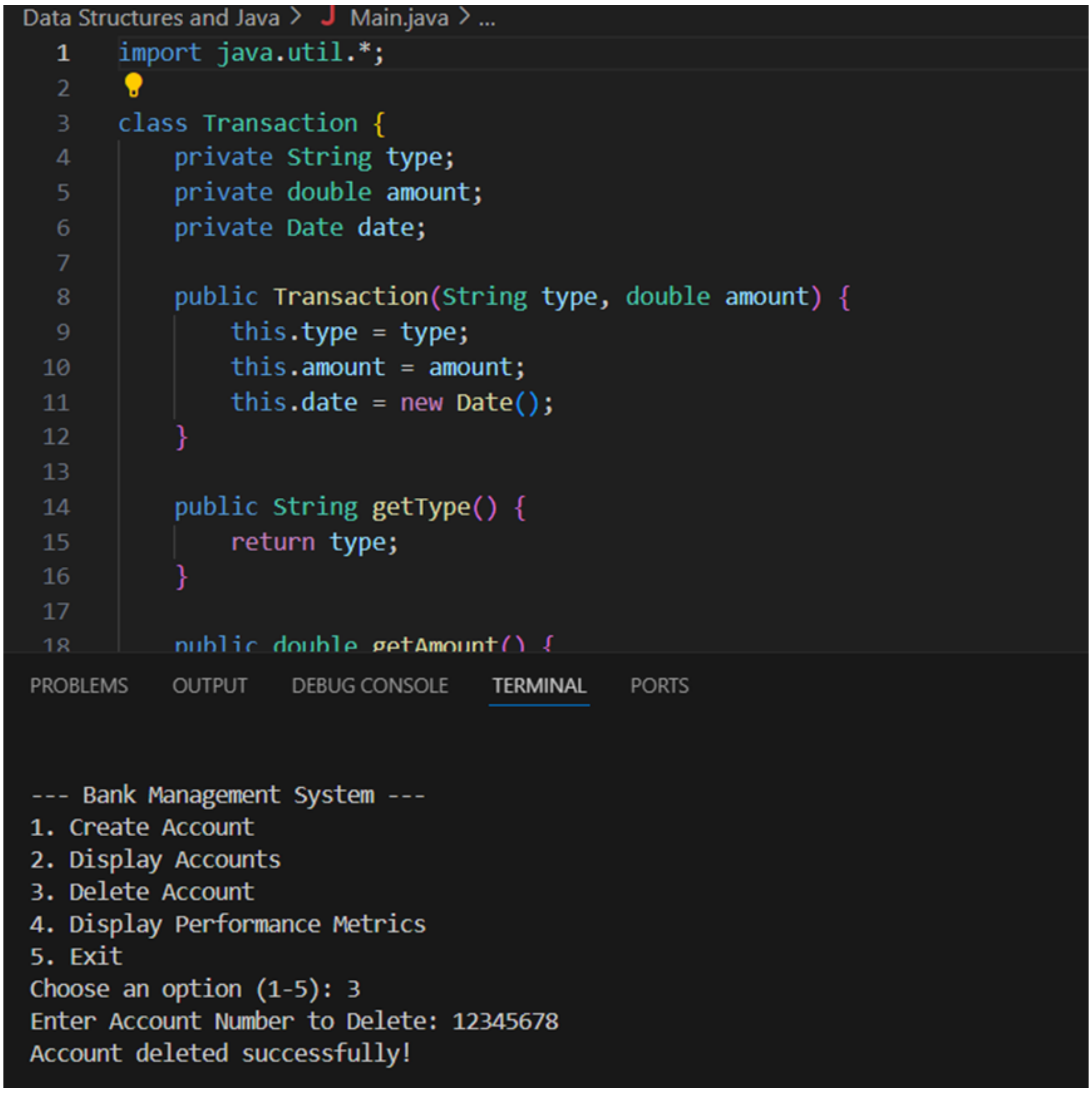

Data Structures and Java > Main.java > ...

```java
import java.util.*;

class Transaction {
    private String type;
    private double amount;
    private Date date;

    public Transaction(String type, double amount) {
        this.type = type;
        this.amount = amount;
        this.date = new Date();
    }

    public String getType() {
        return type;
    }

    public double getAmount() {
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
--- Bank Management System ---
1. Create Account
2. Display Accounts
3. Delete Account
4. Display Performance Metrics
5. Exit
Choose an option (1-5): 2

--- List of Accounts ---
Account Number: 12345678, Name: john, Address: 12 elm streer, Opening Date: Thu Dec 12 22:32:21 GMT 2024, Balance: 500.00
```

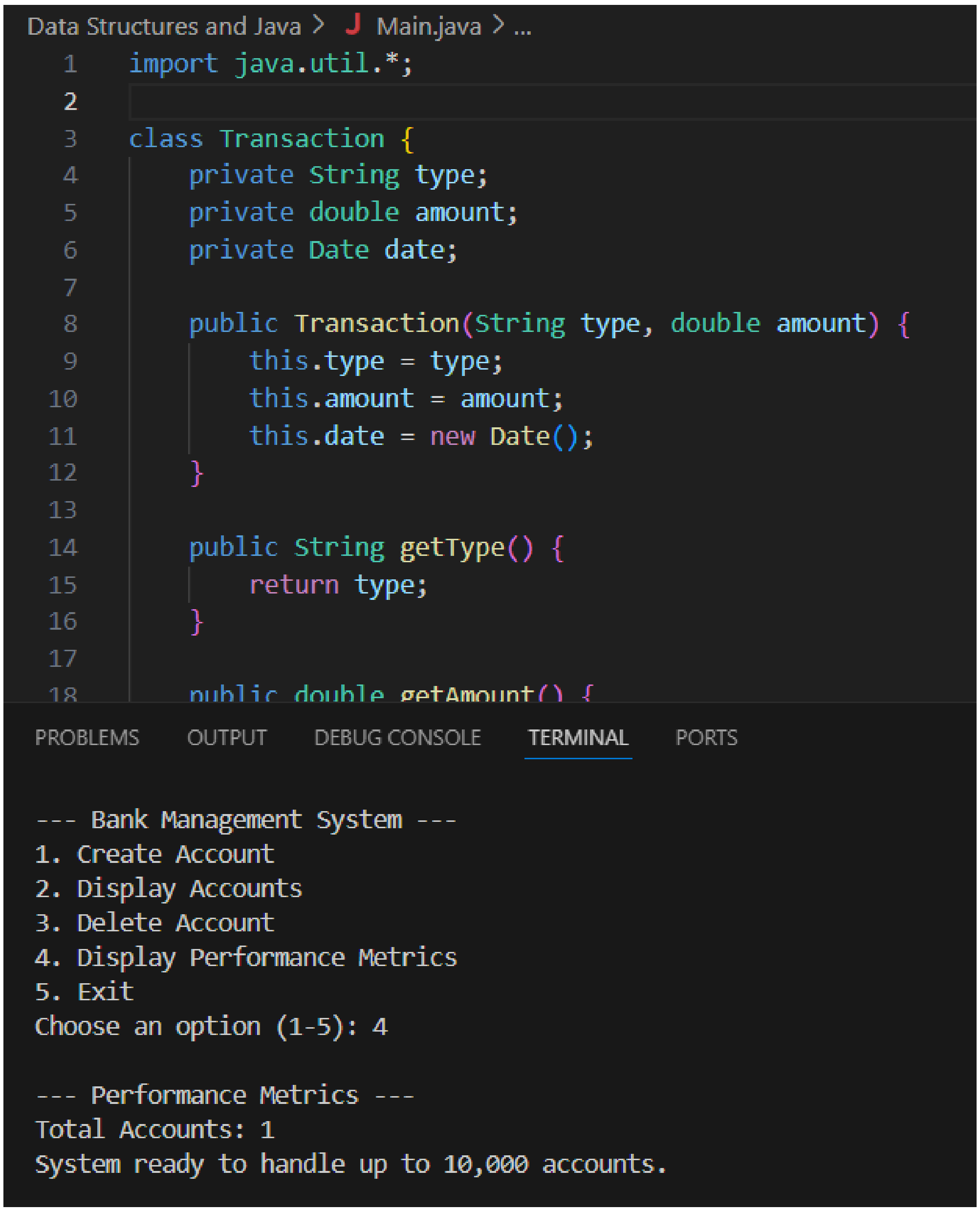
This function lists all created bank accounts along with their details such as account number, holder name, address, and current balance.



```java
import java.util.*;

class Transaction {
    private String type;
    private double amount;
    private Date date;

    public Transaction(String type, double amount) {
        this.type = type;
        this.amount = amount;
        this.date = new Date();
    }

    public String getType() {
        return type;
    }

    public double getAmount() {
```

```
--- Bank Management System ---
1. Create Account
2. Display Accounts
3. Delete Account
4. Display Performance Metrics
5. Exit
Choose an option (1-5): 3
Enter Account Number to Delete: 12345678
Account deleted successfully!
```

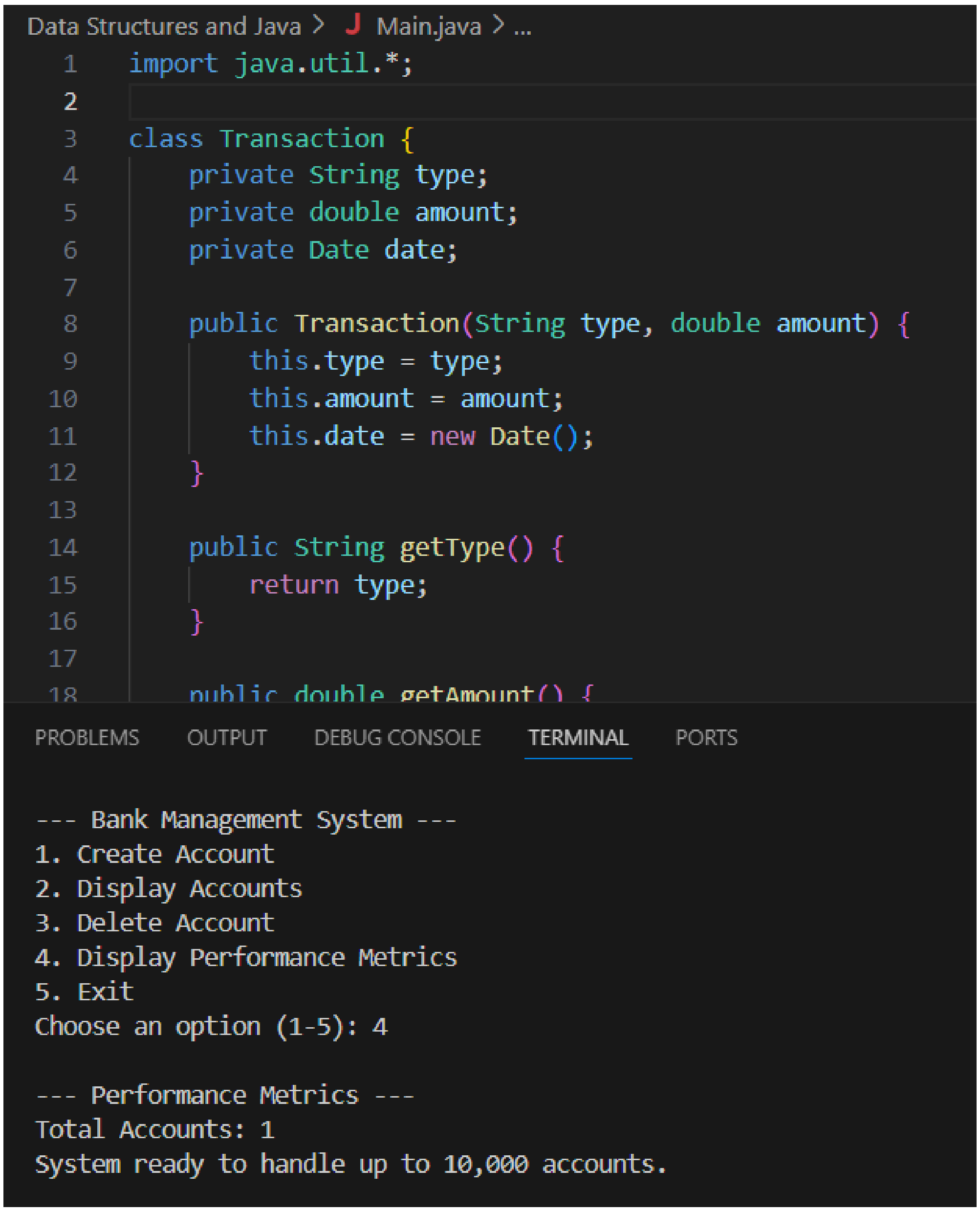
This function allows the user to delete a bank account by providing the account number. It checks if the account exists before deletion.



```java
import java.util.*;

class Transaction {
    private String type;
    private double amount;
    private Date date;

    public Transaction(String type, double amount) {
        this.type = type;
        this.amount = amount;
        this.date = new Date();
    }

    public String getType() {
        return type;
    }

    public double getAmount() {
```

```
--- Bank Management System ---
1. Create Account
2. Display Accounts
3. Delete Account
4. Display Performance Metrics
5. Exit
Choose an option (1-5): 4

--- Performance Metrics ---
Total Accounts: 1
System ready to handle up to 10,000 accounts.
```

This function shows performance data, including the total number of accounts and a message confirming the system’s capability to handle up to 10,000 accounts.



```java
import java.util.*;

class Transaction {
    private String type;
    private double amount;
    private Date date;

    public Transaction(String type, double amount) {
        this.type = type;
        this.amount = amount;
        this.date = new Date();
    }

    public String getType() {
        return type;
    }

    public double getAmount() {
```

```
--- Bank Management System ---
1. Create Account
2. Display Accounts
3. Delete Account
4. Display Performance Metrics
5. Exit
Choose an option (1-5): 4

--- Performance Metrics ---
Total Accounts: 1
System ready to handle up to 10,000 accounts.
```

This function terminates the program, effectively ending the session and closing the system.

Test Plan, Test Data, and Test Results

Test Plan

The test plan for the Bank Management System aims to verify the core functionalities of the system and ensure it behaves as expected under various conditions. The tests primarily focus on:

Account creation:

Validating correct input formats and ensuring the system rejects invalid data (e.g., incorrect account number length or empty holder name).

Transaction handling:

Ensuring that deposit and withdrawal transactions update the account balance appropriately, and that withdrawal transactions fail when the account has insufficient funds.

Account deletion:

Checking that valid account deletion proceeds correctly and that the system rejects attempts to delete non-existing accounts.

Account display and performance metrics: Ensuring that the system displays all accounts and that the system shows correct performance data regarding account count and readiness.

Test Data

To verify the functionality of the system, a variety of test cases with different input data were used:

Valid Data for Account Creation:

Account Number: 12345678

Holder Name: John Doe

Address: 123 Main St

Initial Balance: 1000.00

Invalid Data for Account Creation:

Account Number: 1234567 (less than 8 digits)

Holder Name: "" (empty name)

Account Number: abcdefgh (non-numeric)

Valid Transaction Data:

Deposit: 500.00

Withdrawal: 300.00

Invalid Transaction Data:

Withdrawal: 2000.00 (more than the current balance)

Test Case for Account Deletion:

Valid Account Number: 12345678

Non-Existing Account: 87654321

Account Creation:

Valid inputs resulted in successful account creation and the display of account details.

Invalid inputs, such as an incorrect account number length or empty holder name, resulted in appropriate error messages being displayed.

Transaction Handling:

Deposit and withdrawal transactions updated the account balance as expected. A successful deposit increased the balance, while a successful withdrawal decreased it.

Withdrawal transactions that exceeded the balance (e.g., attempting to withdraw 2000.00 when the balance was 1000.00) were rejected with an error message stating “Insufficient balance! Transaction failed.”

Account Deletion:

When provided with a valid account number, the account was deleted, and the system confirmed the action.

When a non-existing account number was provided, the system displayed an error message stating “Account not found.”

Account Display:

The “Display Accounts” functionality worked correctly, displaying a list of all accounts with their respective details, such as account number, holder name, address, and balance.

Performance Metrics:

The system correctly displayed performance metrics when prompted, including the total number of accounts and a message stating the system’s readiness to handle up to 10,000 accounts.

In summary, all the key functionalities were tested and validated. The system correctly handled both valid and invalid inputs, ensuring the integrity of operations such as account creation, transactions, and account management. The performance metrics were accurate, and the system was confirmed to be ready for large-scale usage.

Conclusion on reflection of my experience

For this project, I was tasked with developing a Bank Management System, which includes implementing functions including transaction processing, account creation and deletion, and the display of metrics and account data. It was rewarding and challenging for me to have to apply my knowledge of data structures, algorithms, and object-oriented programming during the development process.

One of the main difficulties I faced was developing the system using the appropriate data structures and methods. At first, I wasn't sure how to manage and keep track of transactions for each account efficiently. I considered for a while before deciding to maintain recent transactions in a circular buffer. For me, this was an important learning experience since it demonstrated how to choose the optimal data structure based on the specific requirements of the application. The circular buffer worked well for keeping the most recent transactions to guarantee that memory was used efficiently.

Implementing the binary search method to locate an account using the account number was another difficulty for me. After some study and testing, I was able to effectively incorporate the binary search to expedite the account searching process, but at first I had trouble optimising the search process. I learnt how to use techniques like binary search to make the system work better, especially when there are more accounts.

For the user interface, I decided on a console-based interface, which was a great starting point. However, I struggled to manage user input and ensure that the system processed certain instructions effectively, such as incorrect input or accounts that didn't exist. I overcame this by using error management and input validation to improve user experience and ensure smooth system interaction.

I worked with many courses and integrated them to ensure the system functioned as a whole. It was challenging for me to organise the numerous components, but in the end I created discrete class roles, using the Transaction class to handle transactions and the BankAccount class to manage accounts. This approach helped me stay organised and produce better code.

I learnt the value of planning and testing from this assignment. I found that I could identify edge cases and ensure the system functioned as intended by developing a clear test strategy and running the application using a variety of test data. Additionally, the process enhanced my debugging and error-handling skills.